

SQL and Table API

Date published: 2020-05-15

Date modified:

CLOUDERA

Legal Notice

© Cloudera Inc. 2024. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

Contents

Flink SQL and Table API.....	4
SQL and Table API supported features.....	5
DataStream API interoperability.....	5
Converting DataStreams to Tables.....	6
Converting Tables to DataStreams.....	6
Supported data types.....	7
SQL catalogs for Flink.....	8
Hive catalog.....	8
Kudu catalog.....	9
Schema Registry catalog.....	9
SQL connectors for Flink.....	10
Kafka connector.....	10
Data types for Kafka connector.....	11
SQL Statements in Flink.....	16
CREATE Statements.....	16
DROP Statements.....	17
ALTER Statements.....	18
INSERT Statements.....	18
SQL Queries in Flink.....	19

Flink SQL and Table API

In Cloudera Streaming Analytics, you can enhance your streaming application with analytical queries using Table API or SQL API. These are integrated in a joint API and can also be embedded into regular `DataStream` applications. The central concept of the joint API is a Table that serves as the input and output of your streaming data queries.

There are also two planners that translate Table/SQL queries to Flink jobs: the old planner and the Blink planner. Cloudera Streaming Analytics only supports the Blink planner for Table/SQL applications.

Adding the following Maven dependency to the Flink configuration file allows you to use the Table API with the Blink planner.

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-api-java-bridge_2.12</artifactId>
  <version>1.12.0-csa1.3.0.0</version>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-table-planner-blink_2.12</artifactId>
  <version>1.12.0-csa1.3.0.0</version>
  <scope>provided</scope>
</dependency>
```

SQL programs in Flink follow a structure similar to regular `DataStream` applications:

1. Create a `StreamTableEnvironment` with the Blink planner.
2. Register catalogs and tables.
3. Execute the queries/updates.
4. Execute the `StreamTableEnvironment`.

You can see an example of the structure here:

```
StreamExecutionEnvironment streamEnv = StreamExecutionEnvironment.getExecutionEnvironment();

EnvironmentSettings tableSettings = EnvironmentSettings
    .newInstance()
    .useBlinkPlanner()
    .build();

StreamTableEnvironment tableEnv = StreamTableEnvironment
    .create(streamEnv, tableSettings);

tableEnv.sqlUpdate("CREATE TABLE ...");
Table table = tableEnv.sqlQuery("SELECT ... FROM ...");

DataStream<Row> stream = tableEnv.toAppendStream(table, Row.class);
stream.print();

tableEnv.execute("Print");
```

The Table API exposes different flavors of `TableEnvironment` to the end users that cover different feature sets. To ensure smooth interaction between other `DataStream` applications, CSA only supports using `StreamTableEnvironment`.

StreamTableEnvironment wraps a regular StreamExecutionEnvironment. This allows you to seamlessly go from streams to tables and back within the same pipeline.

You can create StreamTableEnvironment with the following code entry:

```
StreamExecutionEnvironment streamEnv = ...
EnvironmentSettings tableSettings = EnvironmentSettings
    .newInstance()
    .useBlinkPlanner()
    .build();

StreamTableEnvironment tableEnv = StreamTableEnvironment
    .create(streamEnv, tableSettings);
```

When combining regular DataStream and Table/SQL applications, make sure to always call the .execute command on the StreamTableEnvironment instead of the regular StreamExecutionEnvironment to ensure correct execution.

SQL and Table API supported features

The following SQL and Table API features are supported in Cloudera Streaming Analytics.

StreamTableEnvironment (Blink Planner)	Catalogs
- DataStream conversions: - Row - Tuple - Temporary views - SQL update and Query operations - Creating and using catalogs - Listing catalogs, tables, databases and functions	- In-memory - Flink default catalog - Hive - Schema Registry - Kudu
SQL update statements	SQL query statements
- CREATE TABLE - Computed columns - Watermark definitions - Table connectors - Kafka - Kudu - Hive (through catalog) - Data formats (Kafka) - JSON - Avro - CSV - INSERT INTO - VALUES - SELECT queries	- Basic operations - Scan, Project, Filter - Basic aggregations - Group By - Group By Window - Distinct (only windowed) - Joins - Time-windowed stream-stream join - User defined scalar functions (scalar udf)

DataStream API interoperability

The DataStream API interoperability offers you new ways to build your Flink streaming application logic as you can convert the DataStreams to Tables, and the Tables back to Datastreams. This means that you can run SQL queries on your DataStreams. You can also convert the result back to other streams, or insert them into one of the supported table sinks.

The following DataStream type conversions are supported:

- `DataStream<Row>`
- `DataStream<TupleX<...>>`

Converting DataStreams to Tables

When converting DataStreams to Tables you need to define the `StreamTableEnvironment` for the conversion. Cloudera recommends creating the tables with names as it is easier to refer to them in SQL. You should also take the processing and event time into consideration as crucial elements of Flink streaming applications.

`StreamTableEnvironment` is used to convert a `DataStream` into a `Table`. You can use the `fromDataStream` and `createTemporaryView` methods for the conversion. Cloudera recommends that you use the `createTemporaryView` method as it provides a way to assign a name to the created table. Named tables can be referenced directly in SQL afterwards.

Both of these methods take an optional, but recommended, string parameter to define field name mappings. The string must contain a comma separated list of the desired column names. If the string is not specified, the column names are set to `f0`, `f1`, ...`fn`.

```
DataStream<Tuple2<Integer, String>> stream = ...

Table table = tableEnv.fromDataStream(stream, "col_1, col_2");

tableEnv.createTemporaryView("MyTableName", stream, "col_1, col_2");
```

You need to take into consideration the event timestamps and watermarks when converting DataStreams.

The processing time attribute must be defined as an additional (logical) column marked with the `.proctime` property during schema definition.

```
DataStream<Row> stream = ...

Table table = tableEnv.fromDataStream(stream, "col_1, col_2, t
s_col.proctime");
```

For more information on time handling in SQL, see the [Apache Flink documentation](#).

Even time attributes are defined by the `.rowtime` property during schema definition. This can either replace an existing field or create a new one, but in either case, the field holds the event timestamp of the current record.

```
DataStream<Tuple2<Timestamp, String>> stream = ...
stream.assignTimestampsAndWatermarks(...)

Table table = tableEnv.fromDataStream(stream, "event_ts.rowtime,
col_2");
```

For more information on time handling in SQL, see the [Apache Flink documentation](#).

Converting Tables to DataStreams

Tables are updated dynamically as the result of streaming queries. To convert them into DataStreams, you can either append them or retract them based on the SQL query you have chosen.

The `Table` changes as new records arrive on the query's input streams. These `Tables` can be converted back into `DataStreams` by capturing the change of the query output.

There are two modes to convert a `Table` into a `DataStream`:

- **Append Mode:** This mode can only be used if the dynamic `Table` is only modified by `INSERT` changes. For example, it is append-only and previously emitted results are never updated.

- **Retract Mode:** This mode can always be used. It encodes INSERT and DELETE changes with a boolean flag. True marks inserts, and false marks deletes.

Both `toAppendStream` and `toRetractStream` methods take the conversion class or conversion type information as parameters. For the recommended Row conversions, you need to provide the `Row.class`. For Tuple conversions, you need to provide the Tuple `TypeInformation` object manually.

```
Table table = tableEnv.sqlQuery("SELECT name, age FROM People");

DataStream<Row> appendStream = tableEnv.toAppendStream(table, Row.class);
DataStream<Tuple2<Boolean, Row>> retractStream = tableEnv.toRetractStream(table, Row.class);

DataStream<Tuple2<String, Integer>> tupleStream = tableEnv.toAppendStream(
    table,
    new TypeHint<Tuple2<String, Integer>>() {}.getTypeInfo()
);
```



Note: For a detailed discussion about dynamic tables and their properties, see the [Apache Flink documentation](#).

Supported data types

You should review the supported data types before designing your application to have all the information regarding SQL type mappings, timestamp and date types.

Java to SQL type mappings

SQL Type	From DataStream	To DataStream
STRING	String	String
BOOLEAN	boolean/Boolean	boolean/Boolean
BYTES	byte[]	byte[]
DECIMAL(38,18)	BigDecimal	BigDecimal
TINYINT	byte/Byte	byte
SMALLINT	short/Short	short/Short
INT	int/Integer	int/Integer
BIGINT	long/Long	long/Long
FLOAT	float/Float	float/Float
DOUBLE	double/Double	double/Double
MAP	Maps of supported types using MapTypeInfo	Maps of supported types
ARRAY	primitive/object arrays	primitive/object arrays*
ROW	Row	Row

Timestamp and Date types

The Table API supports a wide variety of conversions between `java.sql`, `java.time` and SQL types. For smooth operation, it is recommended to use `java.sql` time classes whenever possible.

SQL Type	From DataStream	To DataStream	
		Tuple	Row
DATE	<code>java.sql.Date</code>	<code>java.sql.Date</code>	<code>java.time.LocalDate</code>

SQL Type	From DataStream	To DataStream	
		Tuple	Row
	java.time.LocalDate*	java.time.LocalDate*	
TIME(0)	java.sql.Time	java.sql.Time	java.time.LocalTime
	java.time.LocalTime*	java.time.LocalTime*	
TIMESTAMP(3)	java.sql.Timestamp	java.sql.Timestamp	java.time.LocalDateTime
	java.time.LocalDateTime (no .row time support)*	java.time.LocalDateTime*	
TIMESTAMP_WITH_LOCAL_TIME_ZONE	java.time.Instant*	java.time.Instant*	java.time.Instant



Note: For information about the limitations regarding DataStream conversion, see the [Release Notes](#).

SQL catalogs for Flink

Cloudera Streaming Analytics supports Hive, Kudu and Schema Registry catalogs to provide metadata for the stored data in a database or other external systems. You can choose the SQL catalogs based on your Flink application design.

For more information about Flink Catalogs, see the [Apache Flink documentation](#).

In-memory catalog

A generic in-memory catalog is enabled by default. However when the in-memory catalog is used, all objects are only available for the lifetime of the session.

Hive catalog

You can add Hive as a catalog in Flink SQL by adding Hive dependency to your project, registering the Hive table in Java and setting it either globally in Cloudera Manager or the custom environment file.

The Hive catalog serves two purposes:

- It is a persistent storage for pure Flink metadata
- It is an interface for reading and writing existing Hive tables

Maven Dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-hive_2.12</artifactId>
  <version>1.12.0-csa1.3.0.0</version>
</dependency>
```

The following example shows how to register and use the Hive catalog from Java:

```
String HIVE = "hive";
String DB = "default";
String HIVE_CONF_DIR = "/etc/hive/conf";
String HIVE_VERSION = "3.1.3000";
HiveCatalog catalog = new HiveCatalog(HIVE, DB, HIVE_CONF_DIR, HIVE_VERSION);
tableEnv.registerCatalog(HIVE, catalog);
```

```
tableEnv.useCatalog(HIVE);
```



Note: According to the latest recommended setup on CDP, the Hive service hosts only the Hive Metastore Server. Make sure that a Gateway role is installed, or the HMS itself is deployed on the node where the Flink commands are submitted.



Note: Do not use Flink to create general purpose batch tables in the Hive metastore that you expect to be used from other SQL engines. While these tables will be visible, Flink uses the additional properties extensively to describe the tables, and thus other systems might not be able to interpret them. Use Hive directly to create these tables instead.

Kudu catalog

You can add Kudu as a catalog in Flink SQL by adding Kudu dependency to your project, registering the Kudu table in Java, and enabling it in the custom environment file.

The Kudu connector comes with a catalog implementation to handle metadata about your Kudu setup and perform table management. By using the Kudu catalog, you can access all the tables already created in Kudu from Flink SQL queries.

The Kudu catalog only allows to create or access existing Kudu tables. Tables using other data sources must be defined in other catalogs, such as in-memory catalog or Hive catalog.

Maven Dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kudu_2.12</artifactId>
  <version>1.12.0-csa1.3.0.0</version>
</dependency>
```

The following example shows how to register and use the Kudu catalog from Java:

```
String KUDU_MASTERS="host1:port1,host2:port2"
KuduCatalog catalog = new KuduCatalog(KUDU_MASTERS);
tableEnv.registerCatalog("kudu", catalog);
tableEnv.useCatalog("kudu");
tableEnv.listTables();
```

For more information about the Kudu connector and catalog, see the [official documentation](#).



Note: For information about the limitations regarding Kudu catalog, see the [Release Notes](#).

Schema Registry catalog

The Schema Registry catalog allows you to access Kafka topics with registered schemas as Flink SQL tables. You can add Schema Registry as a catalog in Flink SQL by adding the dependency to your project, registering it in Java, and enabling it in the custom environment file.

Each Kafka topic will be mapped to a table with TableSchema that matches the Avro schema.

Maven Dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-avro-cloudera-registry</artifactId>
  <version>1.12.0-csa1.3.0.0</version>
</dependency>
```

The following example shows how to register and use the Schema Registry catalog from Java:

```
SchemaRegistryClient client = new SchemaRegistryClient(
    ImmutableMap.of(
        SchemaRegistryClient.Configuration.SCHEMA_REGISTRY_URL.name(),
        "http://<your_hostname>:7788/api/v1"
    )
);
Map<String, String> connectorProps = new Kafka()
    .property(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
        "<your_hostname>:9092")
    .startFromEarliest()
    .toProperties();

tableEnv.registerCatalog(
    "registry", new ClouderaRegistryCatalog("registry", client, connectorPr
ops)
);
tableEnv.useCatalog("registry");
```



Note: The SSL related properties have to be set independently for both the SchemaRegistryClient and Kafka.



Note: For information about the limitations regarding Schema Registry, see the [Release Notes](#).

SQL connectors for Flink

In Flink SQL, the connector describes the external system that stores the data of a table. Cloudera Streaming Analytics offers you Kafka and Kudu as SQL connectors. You need to further choose the data formats and table schema based on your connector.

Some systems support different data formats. For example, a table that is stored in Kafka can encode its rows with CSV, JSON, or Avro. The table schema defines the schema of a table that is exposed to SQL queries. It describes how sources and sinks map the data format to the table schema.

Kudu connector

The Kudu connector in Cloudera Streaming Analytics offers compatibility with other supported catalogs, and capability to convert your Kudu tables into DataStreams. You need to add the Kudu dependency to your project, set up the catalog, and you can either use SQL queries or the Kudu catalog directly to create tables.

Kafka connector

Cloudera Streaming Analytics provides Kafka as not only a DataStream connector, but also enables Kafka in the Flink SQL feature. This means if you have designed your streaming application to have Kafka as source and sink, you can retrieve your output data in tables. When using the Kafka connector, you are required to specify one of the supported message formats.

Maven dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka_2.12</artifactId>
  <version>1.12.0-cs1.3.0.0</version>
</dependency>
```

For more information about the Kafka connector, see the [Apache Flink documentation](#).

The following example shows a CREATE TABLE statement with Kafka connector:

```
CREATE TABLE source_table (
  id BIGINT,
  ts BIGINT,
  itemId STRING,
  quantity INT
) WITH (
  'connector.type'          = 'kafka',
  'connector.version'      = 'universal',
  'connector.topic'        = 'input_topic',
  'connector.startup-mode' = 'latest-offset',
  'connector.properties.bootstrap.servers' = '<hostname>:<port>',
  'connector.properties.group.id' = 'test',
  'format.type' = 'json'
);
```

On a secured environment where Kerberos and SSL is enabled, the following example can be used:

```
CREATE TABLE source_table (
  c1 STRING
) WITH (
  'connector.type'          = 'kafka',
  'connector.version'      = 'universal',
  'connector.topic'        = 'source_topic',
  'connector.startup-mode' = 'earliest-offset',
  'connector.properties.bootstrap.servers' = '<host>:<port>',
  'connector.properties.group.id'        = 'test',
  'connector.properties.security.protocol' = 'SASL_SSL',
  'connector.properties.sasl.kerberos.service.name' = 'kafka',
  'connector.properties.ssl.truststore.location' =
  '<absolute_path_to_jks>',
  'format.type'          = 'csv'
)
```



Important: The Kerberos related properties should be passed to the run command when submitting your job:

```
flink run -m yarn_cluster -d \
-yD security.kerberos.login.principal=<principal> \
-yD security.kerberos.login.keytab=<local_path_to_keytab> \
-c com.cloudera.streaming.examples.flink.KafkaSecureIT \
flink-sql-tests-1.0-SNAPSHOT.jar
```

Data types for Kafka connector

When reading data using the Kafka table connector, you must specify the format of the incoming messages so that Flink can map incoming data to table columns properly.

JSON format

The JSON format enables you to read and write JSON data. You must add the JSON dependency to your project and define the format type in CREATE table to JSON.

The expected JSON schema will be derived from the table schema by default. Specifying the JSON schema manually is not supported.

Maven dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-json</artifactId>
  <version>1.12.0-csa1.3.0.0</version>
  <scope>provided</scope>
```

```
</dependency>
```

For more information about the JSON format, see the [Apache Flink documentation](#).

The following example shows the Kafka connector with JSON data type:

```
CREATE TABLE source_table (
  c1 INT,
  c2 STRING,
  c3 DECIMAL(38,18),
  c4 TIMESTAMP(3),
) WITH (
  'connector.type'          = 'kafka',
  'connector.version'       = 'universal',
  'connector.topic'         = 'input_topic',
  'connector.startup-mode'   = 'latest-offset',
  'connector.properties.bootstrap.servers' = '<hostname>:<port>',
  'connector.properties.group.id' = 'test',
  'format.type' = 'json'
);
```

CSV format

The CSV format allows your applications to read data from, and write data to different external sources in CSV. You must add the CSV dependency to your project and define the format type in CREATE table to CSV.

The CSV format will derive format schema from the table schema by default. The format schema can be defined with Flink types also, but this functionality is not supported yet.

Maven dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-csv</artifactId>
  <version>1.12.0-cs1.3.0.0</version>
  <scope>provided</scope>
</dependency>
```

For more information about the JSON format, see the [Apache Flink documentation](#).

The following example shows the Kafka connector with CSV data type:

```
CREATE TABLE source_table (
  c1 INT,
  c2 STRING,
  c3 TIMESTAMP(3)
) WITH (
  'connector.type'          = 'kafka',
  'connector.version'       = 'universal',
  'connector.topic'         = 'sink_topic',
  'connector.properties.bootstrap.servers' = '<host>:<port>',
  'connector.properties.group.id' = 'test',
  'format.type' = 'csv'
)
```

Avro format

The Apache Avro format enables you to read and write Avro data. You must add the Avro dependency to your project and define the format type in CREATE table to Avro. You also need to specify the fields of the Avro record within the table.

The format schema can be defined either as a fully qualified class name of an Avro specific record or as an Avro schema string. If a class name is used, the class must be available in the classpath during runtime.

When using the Avro schema string, you must specify the fields of the Avro record. The schema must correspond to the schema of the table in Flink.

For a detailed description of Avro schemas, see the [Apache Avro documentation](#).

Maven dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-avro</artifactId>
  <version>1.12.0-cs1.3.0.0</version>
  <scope>provided</scope>
</dependency>
```

The following example shows how to use an Avro schema string when creating a Kafka connector table. It is specified as a JSON object, having record as type, a name, and the specification of its fields. Note the correspondence of various data types, especially the decimal and array fields.

Example

```
CREATE TABLE source_table(
  string_field STRING,
  long_field   BIGINT,
  decimal_field DECIMAL(38,18),
  int_arr_field ARRAY<INT>
) WITH (
  'connector.type'           = 'kafka',
  'connector.version'       = 'universal',
  'connector.topic'         = 'input_topic',
  'connector.properties.group.id' = 'test',
  'connector.properties.bootstrap.servers' = '<hostname>:<port>',
  'format.type'             = 'avro',
  'format.avro-schema' =
    '{
      "type": "record",
      "name": "test",
      "fields": [
        {"name": "string_field", "type": "string"},
        {"name": "long_field", "type": "long"},
        {"name": "decimal_field", "type":
{"type": "bytes",
"logicalType": "decimal",
"precision": 38,
"scale": 18}},
        {"name": "int_arr_field", "type":
{"type": "array",
"items": "int"}}
      ]
    }'
```

Supported basic data types

Before creating your table with Avro format for the Kafka connector, you should review the supported basic data types that you can use in the application.

Avro schema type	Flink data type
string	STRING
boolean	BOOLEAN
bytes	BYTES
int	INT

Avro schema type	Flink data type
long	BIGINT
float	FLOAT
double	DOUBLE



Note: Time, Date, Timestamp are not yet supported.

To specify a field with additional properties, such as the decimal or array fields in the example, the type field must be a nested object which has a type field itself, as well as the needed properties.

An example of a property that must be set this way is a field's logical type. Some types cannot be directly represented by an Avro data type, so they use one of the supported types as an underlying representation. The logical type attribute tells how it should be interpreted. For example, the decimal type – described below – is stored as bytes, while its logical type is decimal.

Decimal type

- Only 38 precision and 18 scale are supported
- Flink data type is DECIMAL(38,18)
- To specify in the Avro schema: {"name": "decimal_field", "type": {"type": "bytes", "logicalType": "decimal", "precision": 38, "scale": 18}}

Array type

Avro allows arrays of supported basic types, except:

- String
- Decimal
- Constructed types (nesting of arrays, rows, maps)

For example, defining an Array of long values:

- In the table definition: arr_field ARRAY<BIGINT>
- Avro schema:

```
{"name": "arr_field", "type": {"type": "array",
  "items": "long"}}
```

Row type

Flink rows can be specified as records in the Avro schema. Fields must be named both in the SQL of the table definition, as well as in the Avro schema string.

- Field names must match between the table declaration and the Avro schema's record description.
- The two name fields in the Avro schema have the following structure:
 - one on the outside is the name of the field
 - one inside is the type object, pertaining to the record definition
- Decimal fields are not supported within rows.
- Rows can be nested, Arrays are also allowed as fields of the Row.

Example table definition:

```
CREATE TABLE source(row_field ROW<f1 INT, f2 STRING, f3 BOOLEAN>)
WITH (...)
```

Corresponding Avro schema:

```
'format.avro-schema' =
  '{
    "type": "record",
    "name": "test",
    "fields": [
      {
        "name": "row_field", "type": {
          "type": "record",
          "name": "row_field",
          "fields": [
            {
              "name": "f1", "type": "int",
            },
            {
              "name": "f2", "type": "string",
            },
            {
              "name": "f3", "type": "boolean",
            }
          ]
        }
      }
    ]
  }'
```

Map type

Only Maps with String keys are supported. The value field can be any type of the supported ones, except decimal.

- In the table definition: `map_field MAP<STRING,BIGINT>`
- In the Avro schema:

```
{
  "name": "map_field", "type": {
    "type": "map",
    "values": "long"
  }
}
```

Nullability

To set a field nullable in the Avro schema, create a union of the field's type and null. A nullable integer field would be defined as: `{ "name": "int_field", "type": ["int", "null"] }`

Schema Registry Avro format

You can avoid defining the Avro schema for Kafka table sources and sinks, when the schema is stored in Cloudera Schema Registry.

Such topics are accessible through automatically generated tables from the read-only registry catalog.

Maven dependency

```
<dependency>
  <groupId>org.apache.flink</groupId>
  <artifactId>flink-avro-cloudera-registry</artifactId>
  <version>1.12.0-csa1.3.0.0</version>
</dependency>
```

If you need to define a table outside the registry catalog, the following example can be used:

```
CREATE TABLE source_table (
  id BIGINT,
  name STRING,
  description STRING
) WITH (
  'connector.type' = 'kafka',
  'connector.version' = 'universal',
  'connector.topic' = 'message',
  'connector.startup-mode' = 'latest-offset',
  'connector.properties.bootstrap.servers' = '<hostname>:<port>',
  'connector.properties.group.id' = 'test',
  'format.type' = 'registry',
  'format.registry.properties.schema.registry.url' = 'http(s)://<hostname>:<port>/api/v1'
```

)

Cloudera Schema Registry connector for Flink stores the schema version info in the Kafka messages by default. This means that the `format.registry.properties.store.schema.version.id.in.header` property is set to `false` by default.

The schema name in the registry is usually the same as the Kafka topic name, but can be overridden by the `format.registry.schema-name` property.



Note: The schema used in this case must be already registered in the Schema Registry, otherwise an error will occur.

SQL Statements in Flink

You can use the `CREATE` / `ALTER` / `INSERT` / `DROP` statements to modify objects in the chosen catalog. The statements are executed with the `sqlUpdate()` method of the `TableEnvironment`.

```
...
StreamTableEnvironment tableEnv = StreamTableEnvironment.create(env, settings);
tableEnv.sqlUpdate("CREATE TABLE t1(c1 STRING) WITH (...)");
tableEnv.sqlUpdate("DROP TABLE t1");
...
```

CREATE Statements

You can use `CREATE` statements to register database, table, and function objects into catalogs. You should add the connector, the name of the table, the schema and the data format to the statement based on your application design. You can further customize your table statement with computed columns to reflect time in a Flink application.

For more information about `CREATE` statements, see the [Apache Flink documentation](#).

CREATE TABLE

You can connect the Flink Table API and the Flink SQL programs with external systems to read and write streaming tables. The table declaration is similar to a SQL `CREATE TABLE` statement. The followings can be defined upfront for connecting to an external system:

- Name of the table
- Schema of the table
- Connector
- Data format

You can see an example of the defined parameters:

```
tableEnvironment.sqlUpdate(
    "CREATE TABLE MyTable (\n" +
    "    ...    -- declare table schema \n" +
    ") WITH (\n" +
    "    'connector.type' = '...', -- connector specific properties\n" +
    "    ... \n" +
    "    'update-mode' = 'append', -- declare update mode\n" +
    "    'format.type' = '...', -- format specific properties\n" +
    "    ... \n" +
    ")");
```

Computed column and watermark

A computed column is a virtual column that is generated using the syntax “column_name AS computed_column_expression”. Computed columns are commonly used in Flink for defining time attributes in CREATE TABLE statements.

The WATERMARK defines the event time attributes of a table, and allows computed columns to calculate the watermark in the following form: WATERMARK FOR rowtime_column_name AS watermark_strategy_expression. The expression return type must be TIMESTAMP(3).

```
CREATE TABLE ItemTransactions (
  transactionId BIGINT,
  ts BIGINT,
  itemId STRING,
  quantity INT,
  event_time AS CAST(from_unixtime(floor(ts/1000)) AS TIMESTAMP(3)),
  WATERMARK FOR event_time AS event_time - INTERVAL '5' SECOND
) WITH (
  'connector.type' = 'kafka',
  'connector.version' = 'universal',
  'connector.topic' = 'transaction.log.1',
  'connector.startup-mode' = 'earliest-offset',
  'connector.properties.bootstrap.servers' = '<hostname>:<port>',
  'connector.properties.group.id' = 'test',
  'format.type' = 'json'
);
```

CREATE DATABASE

```
tableEnvironment.sqlUpdate("CREATE DATABASE sample_database");
tableEnvironment.useDatabase("sample_database");
```

CREATE FUNCTION

```
package com.cloudera.udfs;
public static class Hashcode extends ScalarFunction {
  public int eval(String s) {
    return s.hashCode();
  }
}
tableEnvironment.sqlUpdate("CREATE FUNCTION hashcode AS
'com.cloudera.udfs.Hashcode');
```

DROP Statements

You can remove the database, table, and function objects from catalogs with DROP statements. You should also include the IF EXIST statement beside the DROP statement to avoid errors due to non existing objects.

Use IF EXISTS statement to avoid errors on non-existing objects.

For more information about DROP statements, see the [Apache Flink documentation](#).

DROP TABLE/DATABASE/FUNCTION

```
tableEnv.sqlUpdate("DROP TABLE|DATABASE|FUNCTION [IF EXISTS] object_name");
```

ALTER Statements

You can use the ALTER statements to modify already registered databases, tables, and function definitions in the chosen catalogs. There are different limitations for databases, tables and functions when altering them.

For more information about ALTER statements, see the [Apache Flink documentation](#).

ALTER DATABASE

Database properties can be changed, but databases cannot be renamed. You can use DROP and CREATE instead of renaming.

```
tableEnv.sqlUpdate("CREATE DATABASE sample_database");
tableEnv.sqlUpdate("ALTER DATABASE sample_database SET ('key1'='value1')");
```

ALTER TABLE

Tables can be renamed and table properties can also be changed.

```
tableEnv.sqlUpdate("CREATE TABLE sample_table(c1 STRING) WITH ('key1'='value 1')");
tableEnv.sqlUpdate("ALTER TABLE sample_table SET ('key1'='value2')");
tableEnv.sqlUpdate("ALTER TABLE sample_table RENAME TO sample_table2");
```

ALTER FUNCTION

New identifiers, which are full classpath for JAVA/SCALA objects, can be assigned to registered functions.

```
tableEnv.sqlUpdate("CREATE FUNCTION myudf AS 'com.example.MyUdf'");
tableEnv.sqlUpdate("ALTER FUNCTION myudf AS 'com.example.MyUdf'");
```

INSERT Statements

Data can be inserted into sink tables in various ways. You can use the INSERT clause to insert the query result into a table, or you can use the VALUE clause to insert data into tables directly from SQL.

You can use the INSERT clause with different source tables, or the VALUES clause. The following examples show how to use INSERT and VALUES at the same time:

```
tableEnv.sqlUpdate("CREATE TABLE source_table (c1 STRING) WITH (...");
tableEnv.sqlUpdate("CREATE TABLE sink_table (c1 STRING) WITH (...");
tableEnv.sqlUpdate("INSERT INTO source_table VALUES ('foo')");
Table elements = tableEnv.fromDataStream(env.fromElements("bar"));
tableEnv.sqlUpdate("INSERT INTO source_table SELECT f0 from " + elements);
tableEnv.sqlUpdate("INSERT INTO sink_table SELECT * from source_table");
Table table = tableEnv.sqlQuery("SELECT * FROM sink_table");
tableEnv.toAppendStream(table, Row.class).printToErr();
```

For more information about INSERT statements, see the [Apache Flink documentation](#).



Note: The following composite types are supported:

- To construct an array in the SQL insert statement: `ARRAY[1,2,3]`
- To construct a row: `ROW[1,'asd',1.2]`
- To construct a map: `MAP['key1','val1','key2','val2']`
- If you get an error such as `Unsupported cast from 'MAP' to 'MAP'`, (or `ROW` to `ROW`, `ARRAY` to `ARRAY`), it can be because the inner types are not matching. Make sure that when inserting, you properly cast the values that build up the composite type. Example: `INSERT INTO t VALUES (map['www', cast(13 as bigint)])`

SQL Queries in Flink

A Table can be used for subsequent SQL and Table API queries, to be converted into a DataSet or DataStream, and to be written to a TableSink. You need to specify the SELECT queries with the `sqlQuery()` method of the `TableEnvironment` to return the result of the SELECT query as a Table.

SQL and Table API queries can be seamlessly mixed, and are holistically optimized and translated into a single program.

In order to access a Table in a SQL query, it must be registered in the `TableEnvironment`. A Table can be registered from the following ways:

- `TableSource`
- `Table`
- `CREATE TABLE` statement
- `DataStream`
- `DataSet`

Alternatively, users can also register catalogs in a `TableEnvironment` to specify the location of the data sources.

The following is an example of SQL query in Java:

```
DataStream<Tuple2<String, Integer>> transactionStream = ...
tEnv.createTemporaryView("Transactions", transactionStream, "account, amount");

Table balance = tEnv.sqlQuery(
    "SELECT account, sum(amount) as balance FROM Transactions GROUP BY account");

DataStream<Tuple2<Boolean, Row>> balanceStream = tEnv.toRetractStream(balance, Row.class);
```

For the detailed documentation and the example code for the different query types, see the [Apache Flink documentation](#).



Note: For information about the supported queries in SQL and Table API, see the [Supported Features](#).